Advocate: A Distributed Architecture for Speech-to-Speech Translation

A. Ryan Aminzadeh and Wade Shen

Advocate is a set of communications application programming interfaces and service wrappers that serve as a framework for creating complex and scalable real-time software applications from component processing algorithms. Advocate can be used for a variety of distributed processing applications, but was initially designed to use existing speech processing and machine translation components in the rapid construction of large-scale speech-to-speech translation systems. Many such speech-to-speech translation applications require real-time processing, and Advocate provides this speed with low-latency communication between services.

In recent years, speech translation technologies have advanced out of research institutions and into real-world applications. As these technologies have become more accurate (see the companion article "Machine Translation for Government Applications," on page 41), many applications once requiring human translators have now become feasible using automatic methods.

Much of the improvement in the last decade has been driven by investment in these technologies by the Defense Advanced Research Projects Agency (DARPA). Currently, DARPA is sponsoring two programs: Global Autonomous Language Exploitation (GALE) and TRANSTAC, which stands for Spoken Language Communication and Translation System for Tactical Use. Both programs have a strong speech translation focus, and both have created real, working machine translation (MT) systems for military applications. These programs differ in their application: GALE is building large-scale speech translation systems for foreign-media-monitoring applications, whereas TRANSTAC is concerned with building speech-to-speech translation systems for smaller-scale (often in embedded systems) tactical communications.

In both cases, the core technologies are the same: speech recognition and MT. Currently, the U. S. Army is transitioning these technologies developed under DARPA into a large-scale acquisition program called Sequoyah. As part of this program, the Army will be developing systems to support platform-independent speech-to-speech (S2S), speech-to-text, and text-to-text translation that use components supplied by algorithm vendors. Therefore, a framework is needed that allows integrators to combine best-of-breed speech and translation technologies for platforms ranging from a single handheld device to a large computer farm. As the program is far-reaching in its objectives and timeline, the applications that are currently envisioned are likely to change. This means that a framework for Sequoyah should be flexible enough to allow support for many different kinds of applications.

The framework developed here is called Advocate, a set of communications application program interfaces (APIs) and service wrappers that enable existing software components to be combined into a single application and run either on a local machine or across a distributed grid of computers. Advocate provides an infrastructure for system designers to combine different computationally intensive processing algorithms (for instance, speech recognition and MT) that can either run on a single processor or a distributed grid of computing resources using the same API. Component algorithms in the framework are wrapped to create services that can run persistently so as to avoid repeated startup.

Advocate uses existing standards to allow platform-independent communication and automatic resource management and discovery, making the resulting application easy to configure and use. Compute components in Advocate are wrapped to create services that can then run on a computer grid or as local libraries. Applications are constructed by defining data flow graphs (DFGs) of these services, called advocates. Advocates automate the passage of data between services and collect results.

Advocate was designed to support large-scale speech translation applications that combine speech recognition and MT technologies to process large volumes of speech data. In this article, we describe the design of Advocate, its performance characteristics, and its application in the context of a large-scale speech translation system.

Speech Translation

The typical S2S translation system is a three-stage process. A speech recognition system produces a text transcript from an audio recording, and an MT system converts this text into corresponding text in a different language (e.g., French to English). Finally, a text-to-speech (TTS) synthesis system generates speech based on this translation.

Speech Recognition

State-of-the-art automatic speech recognition (ASR) systems employ hidden Markov models (HMM) to recognize words from an acoustic signal [1]. HMMs are a method of statistical modeling for systems assumed to be Markov processes with unobservable state. An HMM can take a set of observations (the acoustic signal, in this case) and model the probabilities of observations given possible underlying unobservable states (the words uttered in the acoustic signal) and the probabilities of sequences of unobservable states. These models are developed using training data that provides the basis for developing the statistical distributions representing both the observation and transition probabilities.

The search for W*, the translated word or message, (a process called decoding) is performed by taking an acoustic signal (a set of observations) and computing the most likely sequence of underlying words that would have generated the observed acoustic signal. This process combines the probability of a word generating an observation and the probability of a word following a particular sequence of words.

In general, ASR systems can be defined by the following equations:

$$\mathbf{W}^* = \mathrm{argmax}_{\mathbf{W}}\ P(o\,|\,\mathbf{W})P(\mathbf{W})$$

and

$$P(\mathbf{W}) \approx \prod_{i-1}^{N} P(w_i\,|\,w_{i-1}...w_{i-n-1}),$$

where $P(o|\mathbf{W})$ denotes a probabilistic function that models $\mathbf{W}$ to the acoustic observations o (typically called the acoustic model) and $P(\mathbf{W})$ is the probability of generating the current word w given prior words (typically called the language model).

Large-scale ASR systems can make use of millions of parameters to model $P(o|\mathbf{W})$ and $P(\mathbf{W})$. As a result, the search for $\mathbf{W}^*$ can take significant computer resources. Current systems, even heavily optimized, can use more than ten times the real-time computer resources during decoding when maximizing accuracy. Only smaller-scale ASR systems can run on very limited computer resources (e.g., cell phone processors such as the ARM7/9 series). Furthermore, these systems can have a significant startup time (i.e., in model loading and precomputation).

S2S applications can use both large- and small-scale ASR, depending on the target application. For the experimental systems described in subsequent sections, we make use of an internally developed, large vocabulary system as well as several commercial off-the-shelf (COTS) systems. Lincoln Laboratory's Information Systems Technology Group has developed MT systems; in this work, we use them and a variety of COTS systems.

Machine Translation

Currently, the best MT systems use statistical models to learn translations of source words into target words. This process creates a large memorized table of translation pairs, called a phrase table, which maps words from the source language to the target language [2]. During translation, the best hypothesis **T*** is defined by:

$$\mathbf{T}^* = \operatorname{argmax}_T \; P(\mathbf{S}|\mathbf{T})P(\mathbf{T}).$$

As with ASR, $P(\mathbf{T})$ is a language model (typically modeled via n-grams—multiple words that "go together"), and $P(\mathbf{S}|\mathbf{T})$ is the phrase table (or translation model from language **S** to language **T**). The modeling and decoding processes for MT systems, aside from the nature of the data itself, are parallel processes of training and decoding an HMM. In large applications, the phrase table and language model can consist of hundreds of millions or even billions of entries. As a result, the search process used to decode the best translation for a given input sentence can be computationally expensive.

Text-to-Speech Synthesis

Speech synthesizers are used to convert text in the target language into a speech signal. Current systems use either models of speech like those used for ASR (i.e., HMM-based text-to-speech) or explicit waveform chunks from natural speech concatenated together to form the output signal (i.e., concatenative text-to-speech) [3]. In both cases, significant signal processing and search complexity make these systems computationally demanding (although generally less so than ASR or MT). For this work, we employ COTS systems.

All components of an S2S translation system are error prone. That said, different systems don't necessarily make the same errors. System combination attempts to exploit this variation in errors by combining outputs from multiple systems to create an improved hypothesis. In recent years, MT research has focused on ways to either select the most correct hypothesis from multiple systems or integrate hypotheses from multiple systems to create a more correct hypothesis than created by any individual system. Most combination methods rely on consensus between systems to decide correctness.

Existing Frameworks

A number of frameworks have been developed to address the need for building large-scale distributed applications, and S2S translation systems in particular. While these frameworks attempt to address some of the same challenges as Advocate, their focuses diverge from Advocate's in ways described later in this article. Some existing frameworks and their notable features are listed below.

UIMA. Originally developed by IBM, Unstructured Information Management Architecture (UIMA), now distributed open source by Apache, is a large-scale architecture designed for analysis and search [4]. Easily adaptable to speech translation applications, this architecture is designed to be extensible, yet because of its applicability to a wide array of applications, it requires a significant amount of overhead in terms of up-front programming to customize applications. This overhead is necessary to support any sort of information management. Of particular note is the fact that this framework, like the others to be described, operates in discrete units, producing one output for every input. This processing paradigm is not conducive to real-time processing/applications.

Galaxy. Developed by the Spoken Language Systems Group at MIT's Computer Science and Artificial Intelligence Laboratory, Galaxy is a hub-and-spoke style architecture designed specifically for speech applications [5]. Its central hub is programmable and defines the flow of data among servers running computationally expensive speech/language processing components. The central hub, meant to be a lightweight client, also keeps track of the conversation state and history in the speech application. This architecture requires that data be forwarded by the central hub, increasing the overhead associated with data transfer between servers and the hub, or the client, when operating in real time.

OpenAgent. OpenAgent, developed by SRI International, is a research framework that provides software services in a distributed environment [6]. In this framework, services (called agents) register with clients (called facilitators). Using registered agents, facilitators can process requests from users. The framework does not assume that agents will be constantly available (i.e., some may unregister, new agents may join at any time). Thus, facilitators dynamically recompute the set of needed services and the data flow needed to satisfy a user's request.

Unlike Advocate, the data flows between services are not statically defined. Instead a user requests defined "goals" that a facilitator can satisfy using existing agent resources. OpenAgent also differs in that services register with a predefined set of clients (instead of the dynamic service discovery used in Advocate). Resource management and load balancing are not explicitly implemented, though it might be possible to do these processes through extension to the existing framework.

GATE. GATE, or General Architecture for Text Engineering, an open-source infrastructure for human-language technology, was developed at the University of Sheffield [7]. Services, called modules in the GATE architecture, are either wrapped natural-language processing (NLP) components or objects developed from scratch using the architecture's API. GATE is based completely on Java and focuses, much like UIMA, on annotation-based storage and passing of data, implying synchronous, one-in-one-out operation on discrete elements. As of version 2, GATE's processing resources are not distributed or parallel—all execution is done locally. This structure differs from Advocate's distributed, asynchronous, real-time processing structure. GATE also differs from Advocate in its constraints on input/output between modules, which include required byte offsets and adherence to a particular annotation model, as opposed to Advocate's format-agnostic data-passing paradigm.

JAVA/RMI and Jini. Unlike the frameworks listed above, Jini is a general-purpose architecture for distributed computation (i.e., not specifically developed for speech/language applications) [8]. Originally developed by Sun Microsystems for use with Java [9], Jini is an architecture developed for building service-oriented applications in dynamic settings. Like Advocate, Jini supports service discovery and remote access to computer resources (via remote method invocation or RMI). Asynchronous communication can happen in this framework via events.

Jini differs with Advocate in several ways. The basic paradigm that Jini employs allows clients to both access services and act as services themselves for other clients. Furthermore, Jini is restricted in its communications protocols, employing RMI, which is strictly Java-based. Advocate's employment of Extensible Markup Language-Remote Procedure Call (XML-RPC) for communications among services and clients allows it to span across more varied environments [10].

The Advocate Framework

To restate, Advocate is a set of communications APIs and service wrappers that enable existing software components to be combined into a single application with minimal communications and processing overhead. Like the prior frameworks, Advocate was designed to combine existing speech processing components. In contrast to these systems, however, Advocate supports real-time streaming operation and an asynchronous processing model. It also allows for flexible data passing between different processing modules to support various applications; for example, both UIMA-style annotations and Galaxy-style constraint-based routing can be implemented by extending Advocate classes. Furthermore, Advocate is intended to be more portable, easier to use, and scalable to large, high-throughput applications.

The architecture consists of an Advocate client application described by a list of required services, which, in this case, are speech and language processing components, and the required flow of data between them. Services can be configured by the client to return both data and metadata to the client after each stage of processing.

Advocate can work with existing grid/cloud infrastructures, such as Sun Grid Engine or Hadoop. In the case of nonpersistent services (services that are constructed and torn down on a per request basis), scheduling can be done via grid services while interprocess communication is handled by Advocate. When services need to be persistent because of high startup costs, grid resources can be allocated at startup by Advocate.

Dynamic Service Discovery

Advocate provides automatic discovery and registration of its services. To this end, we use Bonjour, a public COTS implementation of the Internet Engineering Task Force Zero Configuration Networking (ZeroConf) group. Bonjour works much in the same way as the domain name service (DNS) once services are registered. Each client runs a special process called the multicast DNS responder that acts as a local registry of existing services. As each service starts, it registers with all available multi-DNS (mDNS) responders via a user datagram protocol (UDP) multicast message that contains its Internet protocol address, port information, a service-type descriptor, and information regarding the server's current load (used by the client to select minimum latency services).

Clients continually browse the registry in their respective mDNS responders for new services that come online. As services join or leave, the registry is updated in real time, as shown in Figure 1. Multiple clients can be deployed at any one time, each with its own unique mDNS responder. Similarly, multiple instances of a given type of service can be running at the same time, each broadcasting its own unique address and/or port number to all the mDNS responders listening for UDP broadcasts. Since the purpose of this architecture is to construct distributed systems built from remote components, the service discovery protocol includes mechanisms for traversing network address translation devices, such as routers, that bridge different private networks. This capability allows for maximum flexibility.

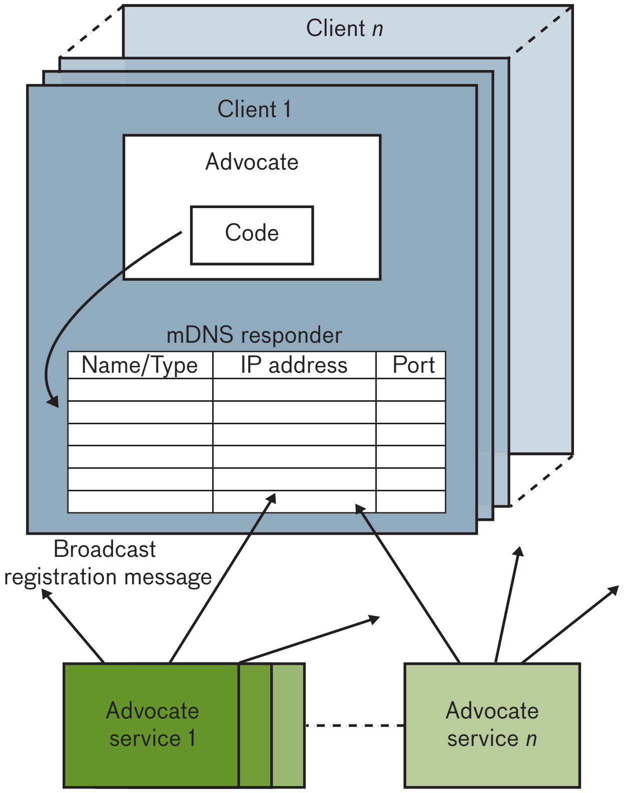


FIGURE 1. Advocate services apply user data protocol broadcast messages to register their name/type, Internet protocol address, and port number with the multi-domain name server (mDNS) responder sitting on the Advocate client's machine.

Clients and Services

In Advocate, clients define a data processing data flow graph (DFG) that consists of multiple services. DFGs can be arbitrarily directed acyclic graphs defining the data flow between services. Using a DFG, each advocate acts as a data processor employing services, ingesting data, and producing output results, along with any by-products that are requested.

Each instance of an Advocate client, shown in Figure 2, is a separate process that runs with a companion mDNS responder (handling the service registry as described above). At their core, Advocate clients implement an XML-RPC client with code to define how the client delegates tasks to services, as well as a `ResultHandler` with an XML-RPC server, which handles asynchronous responses from services once their processing has completed. This module also defines, in code, how processed results are handled. Clients can be easily customized by overriding these modules.

Services in Advocate are data processors. They receive requests from clients (either directly or indirectly forwarded through other services) and process data associated with them. Communications happen in a streaming fashion: (1) clients open streams to services, (2) clients write to that stream, and (3) clients close the stream. Results from services are produced asynchronously and returned to the client via RPC calls to the client's `ResultHandler`. A more detailed Advocate operation procedure follows:

1. Client opens stream to first service.
2. Client sends request to the first service, which writes data to the stream and specifies whether output/metadata ought to be returned to the client when the service produces a result.
3. Service processes data.
4. When a result is generated, the service contacts the client, returning output/metadata if they had been requested. The service then asks whether/where to forward data.
5. The client responds, telling the service whether to forward its data and, if so, the destination.
6. If service is told to forward data, it writes the data to the queue of the next service, opening that service as needed.

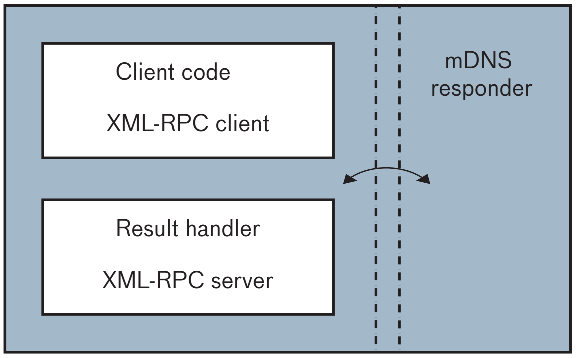


FIGURE 2. An active registry of available Advocate services is maintained throughout, with an XML-RPC client making requests to route data, an XML-RPC server receiving and handling data, and an mDNS responder running as separate processes.

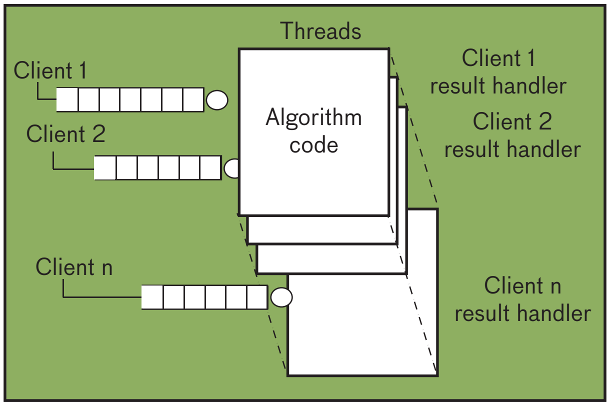


FIGURE 3. An XML-RPC server receiving data and algorithm codes that can be run in multiple threads, each with its own data queue, allows multiple deployments of a service.

Services can handle multiple streams from multiple clients simultaneously. Each stream is associated with a data queue and data processing thread. As data are written to this queue, algorithm code is invoked in the data processing thread. If results and/or metadata are generated, they are queued to corresponding output and metadata queues respectively. This queuing is shown graphically in Figure 3.

Unlike the hub-spoke model (as found in the Galaxy program, for instance), the client does not receive results produced by services unless they are needed. This feature minimizes communications overhead and client load. The overall communication pattern is shown in Figure 4 for a linear, three-service example.

Building Services

Two distinct methods exist for creating Advocate services from compute algorithms:

1. Light service wrappers that run compute algorithms as separate processes. These are useful for rapid prototyping using preexisting binaries and in cases where the source code is not available.
2. Tightly-coupled, link-time library interfaces that call compute algorithms directly.

In both cases, the resulting service appears to have the same interface to clients, and both types of services are available to clients either locally or as distributed services.

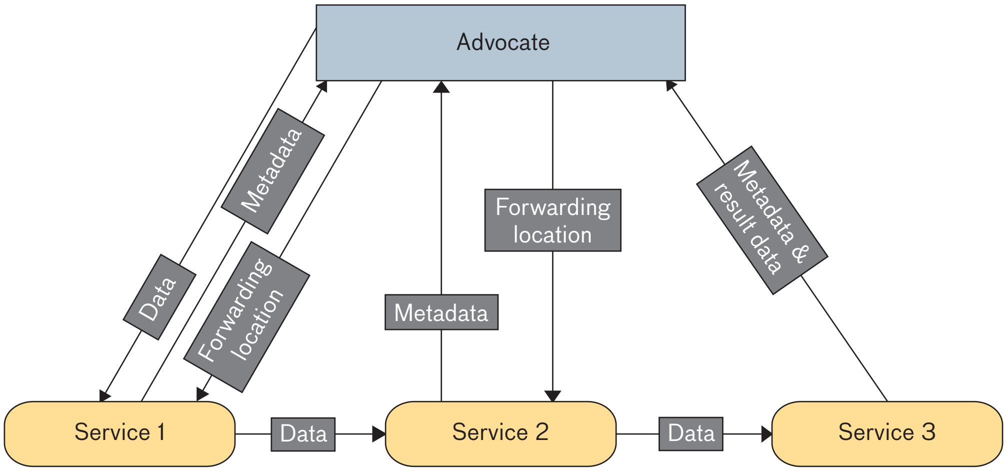


FIGURE 4. Advocate client-to-service communications avoid data transport overhead by having data forwarded between services, allowing the client to act as a mediator, designating where the data should go. Data and metadata are only sent to the client when specifically requested.

```
/**
 * Service Wrapper for MIT/LL MT Service
 */
public class MTService extends AdvocateService {
  /**
   * Main Entry point for this service
   */
  public static void main(String [ ] arg ) throws Exception {
       MTService service = new MTService (Integer to Value(arg[0])) ;
  }
  /**
   * The constructor for local interfacing
   */
  public MTService ( ) {
       super ( )
     // ... do service-specific init for local interfacing
  }
  /**
   * The constructor for service initialization
   */
  public MTService(int xmlrpcPort) throws Exception {
       super ("MTService", xmlrpcPort);
     // ... do service-specific init for RPC interfacing
  }
  /**
   * Thread to process requests (asynchronously)
   */
  private class MTRunServiceThread extends RunServiceThread {
       private MTRunServiceThread (AdvocateRequest request) {
              super(request);
       }
       void runWrapper( ) {
         synchronized(inputLock) {
              byte [ ] inputContent = inputContentQueue.removeFirst( );
         }
         // ... do processing here
         synchronized(outputLock) {
              outputContentQueue.add(output);
              outputMetadataQueue.add(metadata);
         }
       }
  }
}
```

FIGURE 5. Advocate service wrappers are designed to require as little work as possible on the system designer's end. Mechanisms to customize service startup tasks, Remote Procedure Call (RPC) specifications, and component interfacing are built into the code.

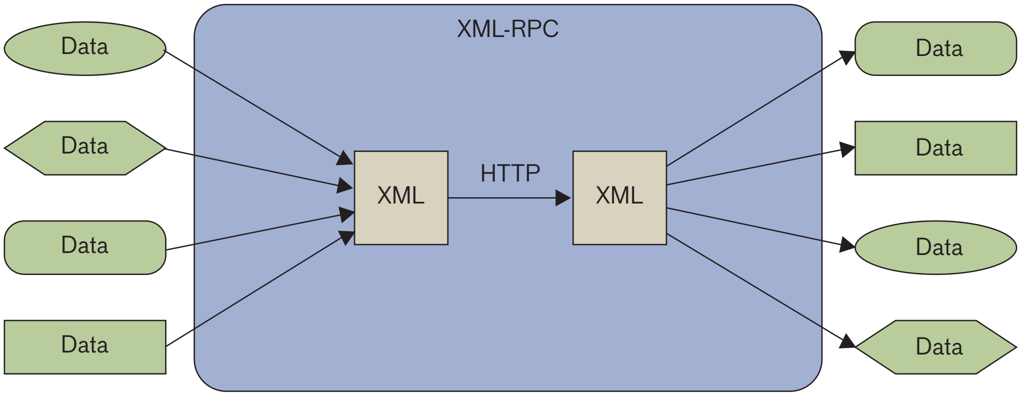


FIGURE 6. XML-RPC allows data to be transferred via remote procedure calls using HTTP. This transfer of data over Transmission Control Protocol gives Advocate its distributed computing capabilities (figure adapted from [10]).

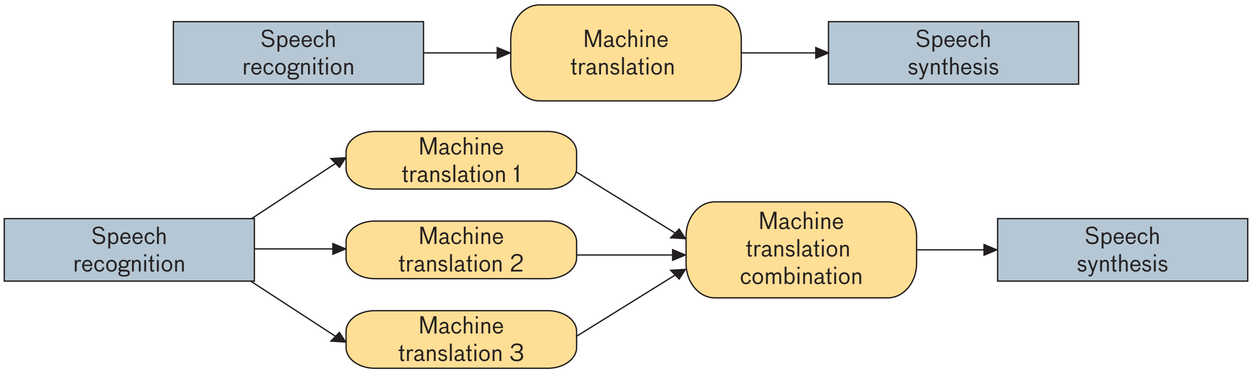


FIGURE 7. In the top example of Advocate applications, there is a linear pipeline for data flow, with each service operating serially on the previous service's results. Shown on the bottom is a corresponding nonlinear pipeline for data flow, where the machine translation (MT) combiner must wait for three MT services, processing in parallel, to complete their processing and forward their results. When there are multiple MTs, the parallel process is more efficient and faster.

Light service wrappers, written in Java, can be used to interface components that are set up and run via command line calls by encapsulating these commands in a structure compatible with Advocate's API. This process is relatively simple, and a template for building these types of services is shown in Figure 5.

Tightly coupled libraries can be used to directly make Advocate API calls from the source code of the component itself. This approach yields maximum flexibility and control over the interface between the component and its service functionality in the context of Advocate, but requires that the source code for the component itself be readily available.

Platform-Independent Communications: XML-RPC

Advocate is designed to be platform neutral with support for communications between clients and services of different architectures. To this end, we use a standard XML-RPC approach, shown in Figure 6, for all communications. XML-RPC uses the hypertext transfer protocol to transport transmission-control protocol messages with XML encoding between computers, allowing for remote procedure calls with complex data-structure transport. Services and clients within the Advocate framework act as XML-RPC servers and clients respectively.

Data Flow Graphs

An Advocate client defines a data flow graph that governs when and how data are passed between services for processing. This is a directed acyclic graph describing the dependencies between different services. Figure 7 shows examples of how services can be configured in Advocate clients. The upper pipeline is a simple S2S application. The lower figure shows a more complicated S2S application in which multiple MT systems are fused to produce a single output.

In the case of S2S translation, often the data flows are linear or near linear (e.g., speech recognition followed by MT). As this pipeline-like structure is quite common, the default implementation of the Advocate client supports a simple XML descriptor interface to describe these pipelines with optional fields where the user can specify which components return their output and any metadata back to the client. An example of this descriptor format is shown in Figure 8.

As previously stated, many existing S2S systems require more complex, nonlinear operational capabilities. Advocate allows the user to, with little effort, override the default linear data flow. In this case, clients, given an arbitrary data flow graph, must compute a topological sort to create a partial order of services, resulting in an ordered list containing sets of services that can be performed in parallel. Each set is then dispatched in order, and results are transmitted between services as defined by the data flow graph. Facilities for sorting, dispatching, and forwarding are built into the Advocate client library and can be customized by the application designer.

Scheduling and Resource Contention

By design, multiple instances of the same Advocate services can be running on a computer farm and accessible

```
<advocate>
<content inputData="/dev/audio-in" outputData="/dev/audio-out"/>
<service destinationClass="SpeechRecognition" returnContent="false" returnMetadata="true"/>
<service destinationClass="MachineTranslation" returnContent="false" returnMetadata="true"/>
<service destinationClass="TextToSpeech" returnContent="true" returnMetadata="true"/>
</advocate>
```

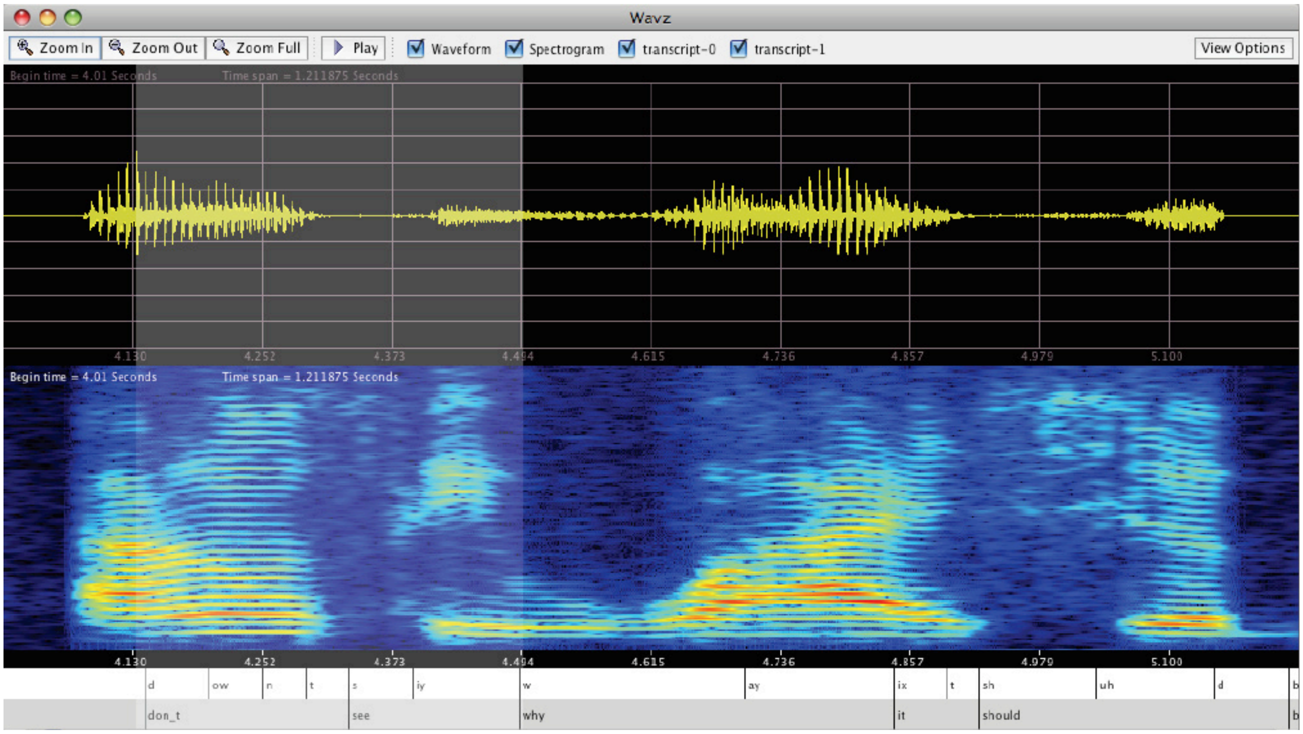
FIGURE 8. This example of an XML file for describing an Advocate application specifies services in the pipeline and optionally indicates points where metadata ought to be returned to the client. This is the default description mechanism for Advocate and applies to linear pipeline descriptions only.

to clients. During startup and while running, services periodically report their loading and status to clients. Clients use this information to determine which services will minimize latency and, by default, they select services using this criterion. Other, more fair, scheduling policies are easy to implement in Advocate as the default behavior can be overridden. Advocate services queue requests from multiple clients, and data processing threads associated with each client are scheduled, by default, on a first-come-first-served basis. Again, this behavior is easily overridden when customizing Advocate services.

Real-Time Streaming and Asynchronous Content

Advocate is designed to support streaming, real-time operation. To this end, data are written to services by clients (and other services) as a stream, and services may produce results asynchronously (i.e., one input may produce zero or many outputs) as described earlier. This approach allows services to produce results as soon as the input has been written (modulo processing latency). As such, the framework does not introduce significant latency in the process.

Consider the example, shown in Figure 9, in which



☑ Waveform ☑ Spectrogram ☑ transcript-0 ☑ transcript-1

Begin time = 4.01 Seconds   Time span = 1.211875 Seconds

FIGURE 9. An asynchronous speech recognition service can generate output without requiring complete processing of the input data. In this case, the highlighted (in gray) speech acts as a single input that generates multiple output words as soon as they are recognized, without waiting for the entire input audio segment to be processed. This configuration allows a single input to generate zero or many outputs.

a client has written the highlighted region of audio (360 ms) comprising the words *don't* and *see* to a speech recognition service. The recognition service can produce the word *don't* as soon as it's able to process the first 160 ms of speech. It will not need to wait until the entire buffer has been processed. In this mode, when a service generates results, it informs the client actively via an RPC callback mechanism (i.e., calling the client's `ResultHandler`) as soon as possible.

Table 1: Advocate Overhead and Computational Latency

AUDIO FILE DURATION (S)	REAL LATENCY (S)	IDEAL LATENCY (S)	MARGINAL LATENCY (S)
4	21.8	21.4	0.4
8	25.8	25.0	0.8
12	31.0	29.8	1.2
16	36.5	35.0	1.5
20	41.92	40	1.92

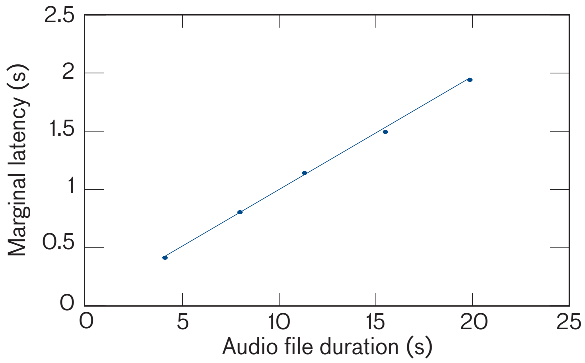


FIGURE 10. The marginal latency associated with Advocate processes is a linear function of the audio file duration. The overhead is approximately 10% of the file size.

Speech-to-Speech Experiments with Advocate

In order to evaluate the performance of the Advocate framework, we put together a demonstration Arabic-to-English speech translation system using both internally developed (Arabic speech recognition and MT) and COTS components (Festival English TTS).

Using this system, we measured the latency incurred by the Advocate framework when compared against the sum of processing latencies for each service's algorithm (i.e., the minimum achievable latency for this pipeline, given the algorithms used with zero-framework latency). We also measured the throughput limit of an Advocate system in terms of its ability to pass data through a data flow graph in which services are running on different computers.

System Latency and Overhead

In order to compute the latency introduced by the Advocate framework, we measured the latency for each of the algorithms in our linear data flow. The ideal latency of such a linear data flow is simply the sum of each component algorithm's processing latency. The difference between the ideal latency and the real latency of a system is the marginal latency. We measured the real latency for five audio files of varying size, since we expected that the marginal latency—the overhead associated with Advocate—should increase with file size. The reason for this expectation is that as file size increases, so does the time required for encoding and decoding XML data transfers from the XML-RPC communications protocol, file-system input and output, and data transmissions across a network.

As we can see in Table 1 and Figure 10, the marginal latency of the Advocate architecture increases linearly with the file size being processed. The marginal latency is a small percentage of the overall processing time when you consider the data sizes being processed. In the real case of a distributed environment with services on different processors, the marginal latency is only about 10% of real-time audio duration. This latency can be further optimized in future versions via compression methods, particularly in the XML encoding schemes used to transmit data among distributed services.

System Throughput

We measured the ability of the Advocate framework to process data in a pipeline fashion under real-world network conditions. The throughput of the system can then be measured by the number of requests the framework can process in a fixed amount of time.

Table 2: Idealized Throughput Rates

AUDIO FILE DURATION (S)	EXPECTED THROUGHPUT (REQUESTS/MINUTE)	ACTUAL THROUGHPUT (REQUESTS/MINUTE)
4	12.0	11.0
8	12.0	10.67
12	12.0	10.5
16	12.0	10.0
20	12.0	9.75

First, we measured Advocate's throughput under idealized conditions by setting up a series of three dummy services (without algorithm processing) that simply transfer input data to the next service in a linear data flow graph after holding the data for a fixed period of 5 seconds each. Five seconds was chosen because it is longer than the delay incurred by the network in transferring the data, and so the latency of the system is guaranteed to be service-process-time bound. Each of the three services ran on separate but identical processor within the same subnet as the Advocate client. Multiple Advocate clients were then configured to send eighty identical requests (audio files of a given size) to the pipeline. The average number of requests processed per minute is then recorded as the throughput.

Under the idealized conditions, we expect that the only delay incurred will be the fixed 5-second delay of the first service, and therefore we would expect a constant throughput of twelve requests per minute. This is because, under these conditions, data of the same size passing among identical services require the same amount of time for processing at each stage; there are no bottlenecks due to data transfers.

However, because of data passing delays across a network, the actual throughput under these conditions will vary with data size. Table 2 shows the results of these experiments and the effects of varying data size on throughput in the idealized case.

Next, we measured Advocate's throughput under the more realistic conditions of our Arabic S2S demonstration system described earlier. Once again, the data size was varied and eighty identical requests were processed, with multiple clients making requests. However, in this case the services are not idealized; therefore, rather than a fixed delay of 5 seconds, the processing delay, and by extension the system's expected throughput, becomes a function of the data size as well. To compute the expected throughput, we averaged the processing time of each component over the eighty trials for a given file size and located the process with the longest delay. This weakest link in the list of services (in this system's case, this was always the ASR) was used as the initial delay in computing how many requests could theoretically be processed per minute. The results of the experiments are shown in Table 3.

As we can see, in both the idealized and actual cases, throughput of the system decreases with the file size and falls short of the expected throughput, given the known latencies of the services for that size of data, whether it be dependent or not. This result indicates that throughput is lowered because of delays incurred during transmission of data between services across a network, and these delays increase when larger amounts of data are being transferred. Performance, however, is fairly close to ideal. What these results show us is that the costs incurred by Advocate are minimal when the processing time of the services is longer relative to the amount of data that need to be transferred. On the other hand, using Advocate in a setting with large amounts of data and very fast services is less than optimal because of the relatively high overhead of data transfer compared to very short service processing times.

Table 3: Realistic Throughput Rates

AUDIO FILE DURATION (S)	EXPECTED THROUGHPUT (REQUESTS/MINUTE)	ACTUAL THROUGHPUT (REQUESTS/MINUTE)
4	3.22	3.17
8	2.46	2.40
12	1.99	1.95
16	1.62	1.59
20	1.38	1.37

Deployment Plans

Internally, Advocate has been demonstrated on clusters of 100+ CPUs. Our testing indicates that latency due to the framework and network communications combined is fairly minimal, and that this framework, therefore, could be scaled to much larger processing grids without increasing network bandwidth.

The Advocate system has been deployed to testbeds at the Air Force Research Laboratory with 200+ CPUs where researchers are evaluating and using a number of COTS and government off-the-shelf (GOTS) ASR/MT/TTS systems for government S2S applications. Advocate combines these systems to create processing data flow graphs that may someday support applications such as coalition communications (e.g., NATO forces communicating cross lingually), document translation, and cross-lingual information search.

The Army's Sequoyah program could potentially use Advocate as a framework for a testbed to evaluate GOTS components. As Lincoln Laboratory has a role in test and evaluation, several prototype S2S translation systems with in-house technologies will be used to design metrics and evaluation procedures. Details of this effort are described in the companion article in this issue.

Other applications of Advocate include large-scale, real-time speech processing for voice search applications. For example, libraries may want to create searchable indices of news broadcasts from multiple simultaneous channels. In these applications, a large computer infrastructure is typically built to run speech recognition technologies and build large database indices for later searching. In this type of application, Advocate can be used to combine speech recognition services with database engines to perform indexing. Multiple Advocate clients can work in parallel to index multiple channels.

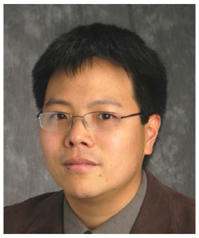
Since Advocate was designed to be a general-purpose framework, many applications requiring distributed, real-time services can be built quickly and efficiently by using nonparallel COTS/GOTS components. In the coming months, we will be working with existing government sponsors to develop some of these applications. As we continue to test and refine Advocate, we intend to release versions to a wider community of developers. ■

REFERENCES

1. L.R. Rabiner and B.H. Juang, "An Introduction to Hidden Markov Models," *IEEE ASSP Mag.*, vol. 3, no. 1, 1986, pp. 4–16.
2. P. Koehn, F. J. Och, and D. Marcu, "Statistical Phrase-Based Translation," *NAACL '03: Proc. 2003 Conf. N. Amer. Ch. Assoc. Comp. Ling. on Human Lang. Tech.*, vol. 1, 2003, pp. 48–54.
3. P. Taylor, A.W. Black, and R. Caley, "The Architecture of the Festival Speech Synthesis," Proc. *Third ESCA/COCOSDA Wkshp. Sp. Syn.*, Blue Mountains, NSW, Australia, November 26-29, 1998, pp. 147–151.
4. D. Ferrucci and A. Lally, "UIMA: an Architectural Approach to Unstructured Information Processing in the Corporate Research Environment," *Nat. Lang. Eng.*, vol. 10, no. 3-4, 2004, pp. 327–348.
5. S. Seneff, E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue, "Galaxy-II: A Reference Architecture for Conversational System Development," Proc. *5th Int. Conf. Spok. Lang. Process. (ICSLP)*, 1998, pp. 931–934.
6. D.B. Moran, A.J. Cheyer, L.E. Julia, D.L. Martin, and S. Park, "Multimodal User Interfaces in the Open Agent Architecture," *Knowl.-Based Sys.*, vol. 10, no. 5, 1998, pp. 295–303: also published in *Proc. 2nd Int. Conf. on Intel. User Interfaces*, 1997, pp. 61–68.
7. H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan, "GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications," *Proc. 40th Ann. Mtg. Assoc. Comp. Ling.*, 2002.
8. http://www.jini.org.
9. http://www.java.com.
10. http://www.xmlrpc.com.

ABOUT THE AUTHORS

A. Ryan Aminzadeh is an associate staff member of the Information Systems Technology Group, where his current areas of research are speaker identification and machine translation. He received his bachelor's and master's degrees in electrical engineering from the University of Pennsylvania. He is a member of Tau Beta Pi and Eta Kappa Nu.



Wade Shen is a staff member of the Information Systems Technology Group. His areas of research involve machine translation and machine translation evaluation; speech, speaker, and language recognition for small-scale and embedded applications, named-entity extraction, and prosodic modeling. Shen received his bachelor's degree in electrical engineering and computer science from the University of California, Berkeley, and his master's degree in computer science from the University of Maryland, College Park. Prior to joining Lincoln Laboratory, Shen helped found and served as Chief Technology Officer for Vocentric Corporation, a company specializing in speech technologies for small devices.